UNITED STATES DISTRICT COURT

FOR THE NORTHERN DISTRICT OF CALIFORNIA

OAKLAND DIVISION

EPIC GAMES, INC.,	)	Case No. 4:20-cv-05640-YGR-TSH
	)	
Plaintiff, Counter-defendant,	)	**WRITTEN DIRECT TESTIMONY**
	)	**OF DR. JAMES W. MICKENS**
v.	)	
	)	The Honorable Yvonne Gonzalez Rogers
	)	
APPLE INC.,	)	Trial: May 3, 2021
	)	
Defendant, Counterclaimant.	)	**Ex. Expert 5**

I. Qualifications

1. My name is James W. Mickens. I am the Gordon McKay Professor of Computer Science at Harvard University, and a co-director of Harvard's Berkman-Klein Center for Internet and Society. At a high level, my research expertise covers the domains of "systems" and "security." **Systems research** investigates methods for designing, building, and testing large, complicated programs. **Security research** explores how to build software and hardware that resist attacks launched by malicious actors. Both areas of research involve careful reasoning about the interactions between software, hardware, and various human actors who leverage computers to pursue benevolent goals or malicious ones.

2. I received a bachelor's degree in computer science from the Georgia Institute of Technology, and a PhD in computer science from the University of Michigan. Post-graduation, I spent seven years as a researcher in the Distributed Systems Group at Microsoft Research. Before coming to Harvard in 2015, I was also a visiting professor at MIT, where I worked with the Parallel and Distributed Operating Systems research group.

3. As a researcher and a teacher, I have extensive experience with the security of distributed systems[1] and mobile systems. For example, my Gibraltar research project[2] examined secure mechanisms for allowing web pages to interact with phone sensors like cameras and GPS units. As another example, my work on the Riverbed project[3] demonstrated how to build datacenter applications that respect user-defined privacy policies for sensitive user data. I have direct programming experience with the Linux operating system and the Windows operating system, having implemented new components in both. The two classes that I teach at Harvard ("CS 161: Operating Systems" and "CS 263: Systems Security") both contain lectures on iPhone security; I have also advised undergraduate research projects involving the security of mobile devices and "the Internet of Things."

[1] A distributed system is a collection of different computers that exchange network messages to achieve a common goal. For example, Facebook is a distributed system—users create posts and read the posts of others by communicating with Facebook machines that live in a Facebook-controlled datacenter. The datacenter machines store user content and coordinate how users interact with that content.

[2] Lin K., Chu D., Mickens J., Zhuang L., Zhao F., and Qiu L (2012). Gibraltar: Exposing Hardware Devices to Web Pages Using AJAX. In *Proceedings of the 3rd Conference on Web Application Development*, Boston, MA, https://www.usenix.org/conference/webapps12/technical-sessions/presentation/lin.

[3] Wang F., Ko R., and Mickens J. (2019). Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation,* Boston, MA, https://www.usenix.org/conference/nsdi19/presentation/wang-frank.

II. Assignment

4. iPhones are portable computing devices made by Apple. Developers that wish to distribute iOS applications ("apps") must submit those apps to a centralized, Apple-managed review process. During the review process, Apple examines a variety of app characteristics, such as functionality (e.g., does the app crash?) and content (e.g., does an app contain objectionable images?). Some parts of the review process also examine the security aspects of an app. Only if an app passes the review process will Apple make the app available to the public via the Apple-run App Store. Apple prevents iPhone users from installing apps from third-party channels. I have been retained by outside counsel for Epic Games, Inc. ("Epic") to evaluate iPhone security as it relates to the App Store and the extent to which iPhone security is independent of the mechanism by which apps are reviewed and later distributed to iPhones.

5. In formulating my opinions in this matter, I have relied on my training and experience in the field of computer science (and more specifically, systems security). I have also relied on public-facing Apple documents, peer-reviewed literature, newspaper articles, and other sources that experts in my field typically and reasonably rely on in formulating their opinions.

III. Summary of Conclusions

6. At a high level, my expert opinion is that iPhone security is in fact significantly independent of the app review process and the distribution channel (however they may be implemented). It is therefore my expert opinion that Apple considerably overstates the security benefits of its own centralized App Store model. As the rest of this document explains in detail, an iPhone's security guarantees could predominantly be enforced by the iPhone's operating system (iOS), not by the Apple App Store. In practice, iPhones do indeed mostly rely on iOS to achieve Apple's stated security goals.

7. I now discuss my conclusions in more detail. My first conclusion is that a computer's operating system (OS) is the entity which manages the activity of that computer. The OS is responsible for orchestrating the interactions between the various programs on that computer, and the various hardware components on that computer which store data, exchange messages with a network, receive inputs from the user, and display results to the user. *See* pp. 3-5, 14.

8. A core responsibility of an OS is to provide security guarantees to a computer's user. The OS does so by restricting the actions that each application can perform. These restrictions limit an application to behaviors that appropriately manipulate data or hardware components belonging to the computer. *See* pp. 5-7.

9. An OS can apply these restrictions to an application regardless of the distribution channel by which the application arrived on the computer. iOS in particular applies security techniques in a way that is agnostic about distribution channels. *See* pp. 5-12.

10. Furthermore, iOS is already capable of installing applications that were not distributed via Apple's App Store (and thus were not reviewed by Apple). This means that Apple

already tacitly acknowledges that OS-based security mechanisms are its fundamental safety enforcers, not the Apple-run app review process. *See* pp. 14-21.

11. Apple's review guidelines do enumerate some security properties that cannot be enforced by an OS alone. However, a variety of empirical evidence suggests that, in practice, Apple's review process does a weak (at best) job of enforcing those additional security guarantees. Thus, my expert opinion is that if Apple allowed iPhone users to opt into app distribution via third-party channels, those users would not suffer from a meaningfully less secure experience. *See* pp. 21-27.

12. Apple allows macOS computers to install third-party apps directly from user-selected third-party distribution channels. This capability is another tacit admission by Apple that the primary guarantor of client-side security is the operating system, not the application distribution mechanism. *See* pp. 27-29.

13. The preexisting app distribution models of iOS and macOS all enforce OS-provided security mechanisms, even though some of these models do not require apps to be centrally reviewed by Apple. These models provide empirical evidence that Apple's OSes are technically capable of supporting third-party app stores that enjoy Apple's standard OS-enforced security guarantees. *See* pp. 29-30.

IV. Overview of Computer Design

14. To understand how OSes provide security for users and their data, one must first understand some basic principles of computer design.

15. My expert opinion focuses on consumer computer devices like smartphones, desktops, and laptops. These kinds of computers have both **hardware** and **software**. Examples of hardware are keyboards and mice (which receive user inputs), touchscreens (which both receive user inputs and display information), and cellular radios (which allow smartphones to communicate with cellular networks). **Central processing units (CPUs)** are a special kind of hardware that can perform mathematical calculations, send commands to other hardware, and receive results from that hardware.

16. To determine which calculations to perform, which commands to send to other hardware, and how to respond to the results, a CPU executes (or "runs") the **instructions** that belong to software. Each instruction forces the CPU to perform a simple computation.

17. A particular instance of software is called an "**application**," "**app**," or "**program**." A program is a list of CPU instructions that perform a specific task when executed.

18. On consumer devices, the programs that users directly interact with are **user-level applications**. Examples of such applications include web browsers like Google Chrome, office productivity tools like Microsoft Excel, and games like Words With Friends. Some of these applications are preinstalled by the vendor who makes or sells the device; others are installed by users after a device has been purchased. Regardless, a consumer device stores its applications on **storage hardware** like a hard drive.

19. To run an application, a computer must first load the application's **instructions** and **data** into hardware called **random access memory (RAM)**. Once the instructions for an application are loaded into RAM, a CPU can read the instructions in RAM and perform the associated activities.

20. If a computer has a single CPU, then at any given time, the computer can only execute the instructions belonging to a single application. To provide users with the illusion that multiple applications are running simultaneously, the CPU repeatedly executes one program for a short time (e.g., 10 milliseconds), and then selects a different program to run for a short time, as shown in the diagram below.

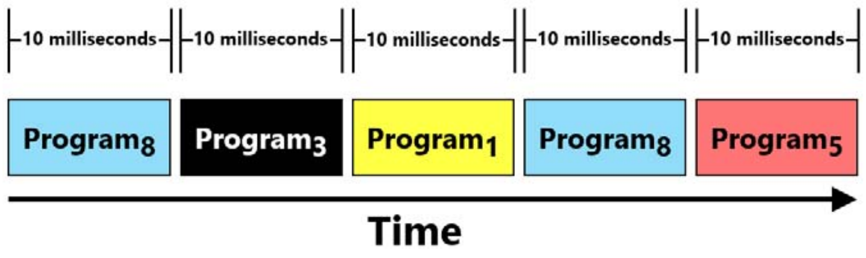

Figure 1: Example of how a CPU rapidly switches between the execution of multiple programs.

From the perspective of a human user, these switches happen so quickly that all of the programs appear to be executing simultaneously. If a computer has multiple CPUs—say, four—the computer can execute up to four applications in parallel at any given moment. Each individual CPU will rapidly cycle between different applications as described in *Figure 1*.

21. A CPU is just one example of a hardware component that must be shared by multiple instances of software. For example, at any given time, RAM and storage devices must contain data from multiple applications; a visual display might need to show information from multiple applications; a Wi-Fi radio might need to be used by two different applications that are streaming video from two different servers. **A key role of OS software is to make decisions about how a computer's hardware is shared between different programs.**

22. A single computer executes programs that were created by a variety of software vendors (e.g., Google, Adobe, Microsoft, and Facebook). Those vendors typically do not trust each other; furthermore, the OS vendor typically does not trust the program vendors. Thus, an important job of an OS is coordinating the activities of the various programs and enforcing security mechanisms which keep those programs from interfering with the proper operation of the computer. For example, on a Windows computer, the Chrome web browser (made by Google) does not trust Photoshop (made by Adobe) to dictate how Chrome loads a web page. Similarly, Photoshop does not trust Chrome to manage how Photoshop edits an image. Thus, Chrome has to trust Windows to prevent Photoshop from interfering with the activity of Chrome; similarly, Photoshop has to trust Windows to safeguard Photoshop from Chrome. Windows cannot delegate enforcement of these security properties to either Chrome or Photoshop, because Windows cannot expect any given program to act in a way that is fair to all other programs on the computer. Indeed, from the perspective of Windows, there is no *a priori* way to eliminate the

possibility that a program is actively harmful. The reason that Windows cannot trust Chrome or Photoshop (or generally, any user-facing application) is that, in general, Microsoft did not write the code for these applications; thus, Windows has no way to verify that the programs are not malicious. This arrangement of trust (and distrust) relationships means that **the OS is responsible for (1) protecting applications from each other and (2) protecting the OS from applications.**

V. How Operating Systems Provide Security

23. OSes employ a variety of important security techniques to prevent a malicious application from interfering with the hardware, other applications, or the OS itself. At a high level, **these techniques remove decision-making power from user-level applications, vesting that power in the OS**. This arrangement forces user-level applications to request the OS's permission to do potentially dangerous activities like write a file or communicate with a network server. The OS can then act as a "security referee," only allowing apps to engage in behaviors that will not hurt the security of the device. **Importantly, all of these security techniques are independent of the mechanism by which user-level applications arrive on a device. In particular an OS would apply these security techniques to an app *regardless of whether that app had been reviewed by a centralized party*.** This observation has important consequences for the safety of third-party app stores (see Section X).

A. Processes and Memory Isolation

24. From the perspective of an OS, a **process** is the fundamental unit of computational activity. A single user-facing application like a web browser is associated with one or more processes. When a user launches a user-facing application (e.g., by clicking on the application's desktop icon), the OS creates a new process for that application, and allocates an initial set of hardware resources. For example, the OS assigns some RAM space to the process. Once the OS has pulled the process's code from the storage device into the RAM space that the OS assigned, the OS instructs the CPU to start executing the process's instructions. *Figure 2* depicts how a single application may consist of several processes.

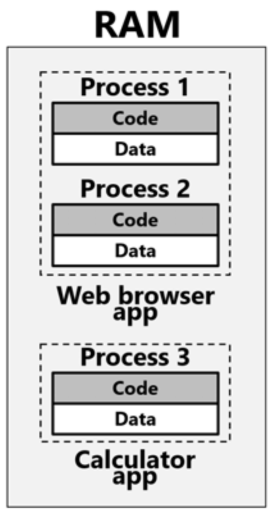

Figure 2: This figure shows a portion of a device's RAM. In this example, there are two user-facing applications running: a web browser and a calculator. The web browser consists of two processes, e.g., because the user has two separate tabs open. In contrast, the calculator only has one process.

After the OS has allocated a RAM area to process X, the OS must ensure that a different process Y cannot directly read the RAM of process X (and thereby potentially steal sensitive information from X). This security property is called **memory isolation**. By restricting the memory locations that a process can directly access, the OS forces cross-process communication to be supervised by the OS, using the system call mechanism that I discuss next.

B. System Calls and Sandboxing

25. As described in Section IV, an OS mediates the interactions between the programs that run on a device, and the hardware components that a device contains. A program uses the **system call** instruction to ask the OS to allow a particular interaction. At a high level, the system call instruction sends a message from a program to the OS, requesting that the OS allow the program to send data to, or receive data from, another program, a hardware device, or the OS itself. When the OS receives the request, the OS determines if the requested interaction should be permitted. For example, a process uses **input/output (I/O)** system calls to communicate with hardware components and other processes.

26. A **system call filter** restricts the kinds of system calls that a process can invoke; a filter may also restrict the kinds of data that an application may pass to the OS via system calls. Remember that system calls are the only way that an application can interact with entities that are external to the application, like the hardware or other applications. Thus, at the extreme, an application that is disallowed from making any system calls at all is effectively prohibited from changing the computer in any way. System call filtering is one of the most important techniques that an OS can use to protect users from malicious or buggy apps. A system call filter is often called a **sandbox**, with a **sandboxed process** being one whose system calls are restricted by a filter. On a smartphone, a sandbox might allow a process to issue system calls that interact with

the storage device; however, the filter might prevent system calls that interact with the phone's cellular radio. In this scenario, a process would be able to read and write files, but not download content from the Internet.

27. Sandboxing allows an OS to enforce **restricted access to all hardware components**, not just RAM, storage devices, and cellular radios. For example, on a smartphone, an OS must control access to sensors like a camera; the OS should prevent an app's processes from taking photos unless the user has explicitly consented to providing camera access to the app.

28. To help OSes enforce these hardware restrictions, modern CPUs provide a distinction between **privileged instructions** and **unprivileged instructions**. A privileged instruction is one that directly manipulates a restricted hardware component, e.g., to send data over a cellular radio. Unprivileged instructions just perform basic mathematical calculations like adding two numbers or dividing two numbers. User-facing applications can only execute unprivileged instructions. In contrast, the OS can execute both privileged and unprivileged instructions. Because, as discussed above, the CPU can only execute one process at a time, the CPU keeps track of whether the OS is currently running, or whether a user-facing process is running. If a user-facing process tries to invoke a privileged instruction, the CPU immediately invokes the OS instead, so that the OS can take the appropriate remediating action (e.g., terminating the misbehaving process). When a user-facing process invokes a system call, the CPU starts executing OS code, allowing that code to directly interact with hardware via privileged instructions.

C. Resource Allocation and Scheduling

29. Even if a process is allowed to access a particular hardware component, the OS must perform **resource allocation and scheduling**. For example, even if a process can access a storage device, the OS must decide how much space each process may consume on that storage device. As another example, the OS's CPU scheduler determines when to let a particular process execute on a particular CPU. As shown in *Figure 1*, a CPU rapidly switches between different processes; when the CPU performs a switch, the CPU asks the OS which process should run next. Similar scheduling decisions arise when multiple processes want to write to a storage device, or multiple processes want to transmit network data using a cellular radio. In each case, the OS decides which order the requests are handled.

30. Consumer-facing OS documentation often uses the broad term "**app isolation**" to express the idea that an OS must restrict how an app interacts with other apps and with hardware components. However, a statement like "the OS must prevent an app from doing ____" is more correctly expressed as "the OS must prevent *any process in the app* from doing ____." In other words, because processes are the fundamental unit of computation, the high-level security goal of an OS is best described as **process isolation**, with the concept of an "app" mostly being useful to define a group of processes that should all be restricted in the same way.

D. Advanced Techniques for Protecting Memory

31. Paragraph 24 described how OSes use memory isolation to stop user-level processes from directly accessing each other's memory regions. This security mechanism is important for reasons of secrecy and integrity. Secrecy means that, by default, process X cannot directly examine sensitive data like passwords that live in the memory of process Y. Integrity means that, by default, process X cannot tamper with the memory belonging to process Y; for example, X cannot modify the code in Y so that Y performs actions that Y's developers did not intend. Secrecy prevents unintended information disclosures, whereas integrity prevents data corruption and behavioral modification.

32. A well-intentioned, non-malicious app developer might nevertheless write code that contains logic errors (i.e., **programming bugs**). Bugs sometimes allow attackers to commandeer the execution of a program, such that the program is forced to perform the attacker's bidding. From the security perspective, some of the most damaging bugs allow attackers to violate memory integrity or memory secrecy.

33. For example, suppose that a phone's text messaging app has a bug involving the way that incoming text messages are processed. When the messaging app's process receives the message from the OS, the app accidentally overwrites part of the app's code with the contents of the text message. This kind of bug allows the attacker to inject new instructions into the process by sending a text message that contains CPU instructions instead of human-readable characters like "hello." When the process executes the injected instructions, the process will act in ways that the attacker controls, in violation of the intentions of the app's developer.

34. To prevent this kind of attack, OSes implement **write exclusive-or-execute memory (abbreviated as W^X memory)**. The basic idea is that the OS separates a process's memory area into pieces, such that a given piece can be updated (i.e., written) or executed (i.e., used to supply the CPU with instructions), but not both. So, in the example from the last paragraph, the OS would allow the text messaging process to have a writable memory region that the process could update to store the incoming text message. However, according to the OS's W^X rules, the writeable memory region could not be used to supply the CPU with instructions. So, if the process tries to ask the CPU to execute the instructions in that memory region, the OS would prevent this from happening. In this manner, even though the attacker was able to *inject* new code into the process, the attacker was unable to get the process to *execute* that code.

35. W^X memory prevents memory integrity attacks like code injection. A different technique called **address space layout randomization (ASLR)** thwarts attacks on memory secrecy. A memory secrecy attack tricks a process into revealing memory data to the attacker. For example, suppose that the text messaging app requires the user to login before actually interacting with the app. The app places the user-inputted password in memory, compares the inputted value with the correct password, and if the two values are the same, the app allows the user to send and receive text messages. When a user types a new message, the messaging app's process stores the message in the process's memory. When the user hits "send," the process should read only the memory locations that store the message, and send only that data over the network. However, suppose that:

- the memory location that stores the message to send is immediately adjacent to the location that stores the user's inputted password, and
- at some point earlier, the process received a maliciously-crafted message which triggered a bug. The bug is that, when the process sends the next outgoing text message, the process will read not just the memory locations that store the outgoing message, but additional memory locations that are adjacent to the outgoing message. Due to the bug, the process sends all of this data over the network. This bug allows the user's password to leak. *Figure 3* demonstrates the problematic memory layout.

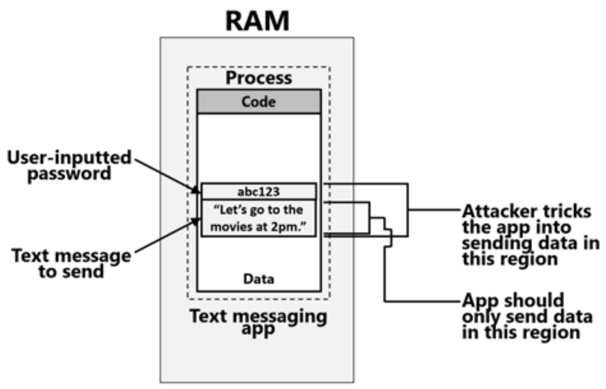
RAM
Process
Code
User-inputted password
abc123
"Let's go to the movies at 2pm."
Text message to send
Data
Text messaging app
Attacker tricks the app into sending data in this region
App should only send data in this region

Figure 3: The memory region belonging to a text messaging app. The data section of the memory region stores the user's password, as well as the next message that the app should send over the network. By triggering a bug in the app, an attacker can trick the app into send "too much data"—namely, the memory data that contain the user's text message, as well as the data in the immediately adjacent location that stores the user's password. The result of triggering this bug is that the user's password is unintentionally revealed to the network (and whoever receives the "too long" text message).

With ASLR, the OS picks a random location to store each kind of memory data belonging to a process. Each time an app is launched, the OS uses a different randomization approach for the processes in the app. In doing so, the OS makes it harder for attackers to profitably exploit program bugs involving memory access. *Figure 4* provides a demonstration.

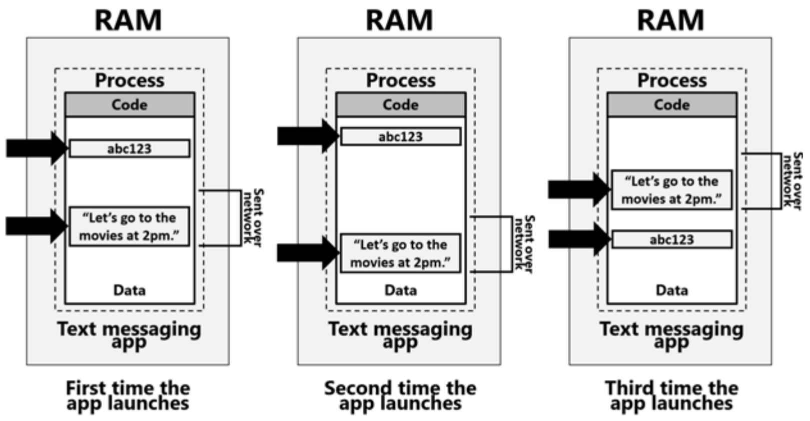
RAM
Process
Code
abc123
"Let's go to the movies at 2pm."
Data
Sent over network
Text messaging app
First time the app launches
Second time the app launches
Third time the app launches

Figure 4: Example of ASLR. Each time the text messaging app is launched, the OS changes the memory locations that store the user's password and the text message that the user wants to send. As a result, even if the attacker can trigger a bug that leaks the memory data adjacent to the message to send, the attacker is unlikely to discover the user's password. In the diagram above, none of the three memory layouts allow the attacker to learn the user's password; the reason is that, in all three layouts, the password lives in a memory region that is not leaked by the bug.

36. Much like a user-level process, an OS has memory regions that contain code and data. Modern OSes use ASLR and W^X to protect their memory pages. Modern OSes also use a variety of approaches to ensure the secrecy and integrity of OS memory with respect to user-level applications who might want to access that memory. For example, iOS's **kernel integrity protection (KIP)** ensures that, once an iPhone completes its startup sequence and is ready to launch user-facing apps, no modifications can be made to the OS's code. To implement KIP, iOS leverages security features provided by the CPU and the memory hardware.

37. KIP stops an attacker from directly overwriting OS code or injecting new OS code. However, the preexisting OS code may still contain bugs that attackers can leverage to trick the OS into disabling the W^X permissions for user-level memory. To prevent such attacks, iOS uses **Page Protection Layer (PPL)** technology. At a high level, PPL separates the OS code into two parts: the part that manages W^X technology, and the rest of the OS. PPL places each part of the OS in a separate, isolated memory region. For example, the region that contains the OS code for W^X management cannot be accessed by user-mode code *or by the OS code that does not manage W^X*. Thus, a security bug in the latter OS code does not automatically result in a compromise of the code that manages W^X. iOS implements PPL using special hardware support available on newer iPhones; the special hardware allows the CPU and memory hardware to implement the intra-OS partitioning with minimal performance degradation. Windows supports a similar hardware-accelerated OS partitioning feature.

E. Secure Booting

38. As described throughout this document, an OS enforces critical security properties for the applications that run on a device. Thus, a secure device must ensure that, when the device is turned on, the device loads (or "boots") an OS which originates from a trusted developer or company. If attackers could force a device to load a malicious or subverted OS, attackers could circumvent the security mechanisms implemented by the legitimate OS.

39. Secure booting is a mechanism for ensuring that, at start-up time, a device boots a legitimate OS. To understand secure booting, one must first understand how **cryptographic signatures** work. A cryptographic signature allows someone (e.g., a person or a company) to vouch for the contents of a digital file. In particular, if Alice signs a file, the signature provides two properties: **attribution** and **integrity**. Attribution means that a third-party Bob can prove that Alice, and only Alice, could have generated the signature on a file. Integrity means that Bob can prove that the signed file was not modified after Alice attached the signature.

40. To sign a file, Alice must generate a **public key** and a **private key**. As the names suggest, Alice can reveal the public key to anyone; however, the private key should only be known to Alice. The two keys are essentially two large numbers that are linked by a special mathematical relationship. The mathematical relationship allows Alice to **sign** a message with her private key. As shown in *Figure 5*, the signature is another large number that Alice attaches to the file that she sends to Bob.

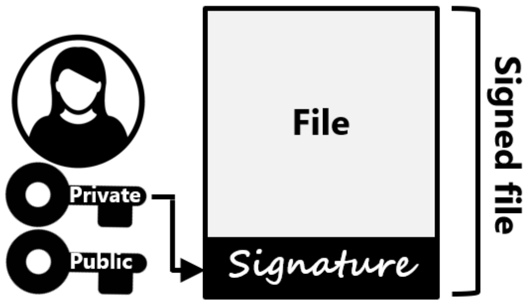

Figure 5: To sign a file, Alice first generates a public key and a private key. Using the private key, Alice generates a signature and attaches it to the file. Later, a different person Bob who knows Alice's public key can download the signed file and vouch for the authenticity and integrity of the file.

When Bob receives the signed file, Bob uses knowledge of Alice's public key to verify that the signature was produced by Alice's private key. The special mathematical relationship between the two keys allows Bob to verify the signature even though Bob does not know Alice's private key.

41. In the context of secure booting, the entity doing the signing is the OS vendor (e.g., Apple or Microsoft), and the entity that verifies the signature is a user's CPU. When a user's device is turned on, the CPU checks whether the OS software that is stored on the storage device has been signed by the OS vendor. If not, the CPU refuses to pull the OS's code from

storage into memory and then execute the OS's instructions. Secure booting therefore allows a computing device to detect if attackers have tampered with an OS (e.g., to disable security features that are implemented by the OS).

VI. OS Design: Kernels vs. Middleware

42. The preceding sections have assumed that an OS is implemented entirely using privileged code (see Paragraph 28). In practice, many OSes employ a different approach, splitting an OS into two parts: a **kernel** which runs privileged code, and **middleware** code that uses unprivileged instructions. This approach allows OS designers to minimize the amount of privileged OS code that is necessary to satisfy an OS's functional requirements. This minimization improves security. The reason is that misbehaving OS code which is privileged can potentially access all hardware and tamper with all processes on a computer; in contrast, misbehaving OS code that is unprivileged can only corrupt its own user-mode process. So, if OS functionality can be implemented in user-mode middleware, OS designers often choose to do so for security reasons.

43. Middleware is often used to provide higher-level, "friendlier" mechanisms for users and developers to communicate with hardware and manage cross-application interactions. For example, consider files that reside on a storage device. An OS kernel provides system calls that allow programs to list the files contained on a storage device. However, human users often want to interact with storage devices visually, using metaphors like "a desktop" and "a file tree." The kernel itself does not provide such a **graphical user interface (GUI)**. However, OS developers can provide a GUI via OS middleware processes. Behind the scenes, the middleware processes invoke the kernel's low-level system calls to interact with the storage device and display results on a monitor or touchscreen. As a concrete example, the Windows kernel uses the File Explorer middleware to provide a GUI to storage devices.

44. There are two kinds of middleware: process-based middleware and library-based middleware.

- **Process-based middleware** employs dedicated, unprivileged processes to mediate how user-facing applications interact with the kernel. In this model, when a user-facing application wants to issue a high-level request to the OS, the application does not use a system call to directly send the request to the kernel. Instead, the application invokes a system call to send the request to the appropriate middleware process.

- In addition to process-based middleware, OSes often provide **library-based middleware**. Library-based middleware is code that a user-facing application should directly incorporate to ease interactions with the kernel. Library-based middleware is commonly used to assist applications that are written in a specific programming language. For example, on Android phones, applications are typically written in the Java language. The Android OS provides Java libraries which mediate communication between the OS and the code that is written by an application's developers; to communicate with the kernel, the developer-written code in the app invokes Java library code, which in turn invokes system calls.

Figure 6 gives an example of the interactions between user-facing applications, library-based middleware, process-based middleware, and the kernel.

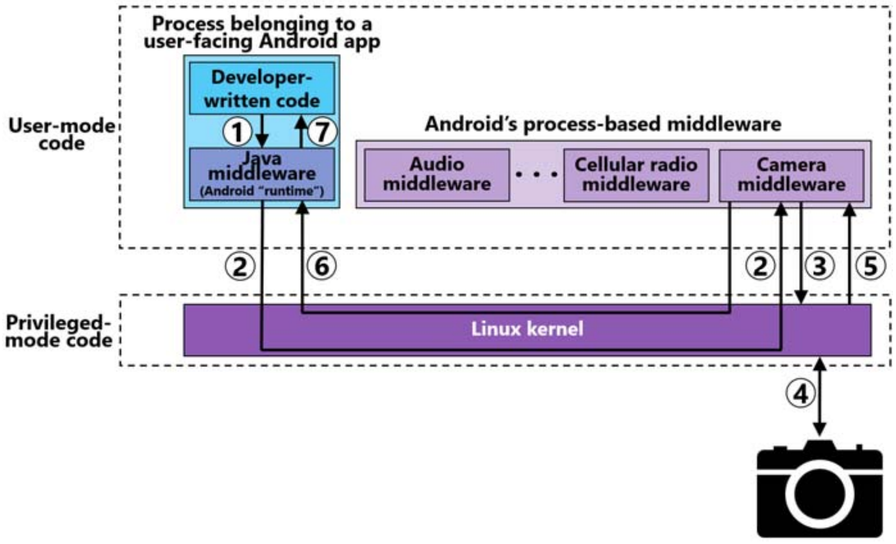
Process belonging to a user-facing Android app
User-mode code
Developer-written code
1
7
Java middleware (Android "runtime")
Android's process-based middleware
Audio middleware
Cellular radio middleware
Camera middleware
2
6
2
3
5
Privileged-mode code
Linux kernel
4

Figure 6: The relationships between developer-written code, library-based middleware, process-based middleware, and the OS kernel on an Android device. In this example, the developer-written code wants to take a photo. The developer-written code invokes functionality provided by the Java middleware library that resides within the app's memory (①). The Java library uses system calls to communicate with Android's process-based middleware that manages camera access (②). The process-based middleware uses system calls to ask the kernel to take a photo (③). The kernel directly accesses the camera, forcing the camera to take a photo and return it to the kernel (④). The kernel returns the photo to the process-based middleware (⑤). The process-based middleware issues a system call to return the photo to the library-based middleware in the app (⑥). Finally, the library-based middleware returns the photo to the developer's code (⑦).

Figure 6 depicts the workflow on Android, but Apple phones use a similar approach. On Apple devices, apps are typically written in Swift or Objective-C; thus, iOS provides middleware libraries for use by Swift apps and Objective-C apps. In the iOS ecosystem, middleware libraries are often referred to as **application programming interfaces (APIs)**.

45. **Application distribution managers** are a common type of process-based middleware. An application distribution manager allows a user to search a catalogue of installable programs and then install one of them, possibly after paying for the application via the manager. Behind the scenes, the manager uses system calls to fetch catalog information over the network, visually display that information, fetch the app data to install, write that data to the local storage hardware, and so on. From the perspective of an iPhone user, Apple's App Store software is an application distribution manager.

46. **Darwin** is the kernel used by both macOS and iOS. Darwin supports the common OS security features that are described in Section V (e.g., memory isolation, mediated process interactions via system calls, ASLR, W^X memory, and secure booting). Note that, even though both macOS and iOS use Darwin as their kernel, macOS and iOS do not share all of their process-based middleware. For example, both macOS and iOS include `launchd`, which is process-based middleware that manages other instances of process-based middleware. On both macOS and iOS, `launchd` creates process-based middleware to handle GUI interactions. However, smartphones receive inputs and display results using a touchscreen, whereas laptops and desktops usually interact with users via mice, keyboards, and non-touchscreen displays. Thus, `launchd` creates different GUI middleware for macOS and iOS. On iOS, `launchd` creates `SpringBoard`, which provides touchscreen support; in contrast, on macOS, `launchd` creates `Finder`, which supports mice, keyboards, and traditional monitors.

VII. The iOS Operating System

47. iOS allows apps to be installed in various ways.

- The best-known method is via the Apple-controlled **App Store**. For most iPhone owners, the App Store is the only Apple-sanctioned mechanism for procuring apps. The App Store model prohibits developers from directly distributing apps to users. Instead, developers submit their apps to the Apple-managed **App Review process**. Only if an app passes the review will Apple make the app available via the App Store. Each app on the App Store is signed by Apple, to indicate that Apple has indeed reviewed the app and declared the app to satisfy the review guidelines.

- Apple allows businesses to distribute apps directly to their employees' phones using the **Developer Enterprise Program**. In this distribution method, apps are signed by enterprise developers, not by Apple; furthermore, apps are not reviewed by Apple.

- Apple enables yet a third distribution mechanism for Apple engineers, allowing those engineers to load **unsigned development apps** as the engineers test iOS features. These apps are unreviewed by the centralized Apple review process, and are not signed by anyone.

In the following subsections, I describe each of these distribution models in more detail. However, I note here that, **for each of these models, iOS enforces the security mechanisms described in Section V (e.g., memory isolation, ASLR, etc.). The reason is that these OS-enforced security mechanisms are independent of the app distribution model used by an iPhone.**

A. The App Store

i. *App Review Guidelines*

48. Apple's public-facing **Review Guidelines** enumerate five high-level metrics that Apple uses to evaluate apps: "safety," "performance," "business," "design," and "legal." The metric names are somewhat misleading, because the app characteristics that Apple examines for each metric are broader in scope than what the metric name would suggest. For example, Apple inspects an app's security characteristics during the "performance" and "legal" checks, not just during the "safety" checks.

49. I will primarily focus on the security-related properties that are mentioned by Apple's review guidelines. I define a security property as one that, if not enforced, would:

- make an app easier to subvert,
- allow an app to improperly interact with other apps, the iOS operating system, a device's hardware components, or
- expose sensitive user data to potential theft or corruption by that app or by other apps installed on the phone.

Apple's review guidelines mention a variety of non-security app characteristics which I broadly refer to as **"quality assurance" (QA)** properties. QA properties reflect judgments about what makes user interactions with an app more enjoyable (e.g., because the app's visual design is aesthetically pleasing, and the app does not consume battery power too quickly). "Enjoyable" is defined from the perspective of Apple, not from some universally-accepted understanding of QA properties. An app with poor QA properties may still be secure.

50. Apple's developer-facing documentation suggests that the review process checks apps for five security properties:

- **Sandbox compliance** means that an app is restricted in how it interacts with external entities like the OS, the local hardware, and other apps. If an app is sandboxed, then the amount of damage that the app can unleash is limited if that app is intentionally malicious or is corrupted by a malicious actor.
- **Exploit resistance** indicates that a user-level application is difficult to corrupt by a malicious actor. Exploit resistance also means that a user-level application cannot behave in a manner which is likely to corrupt the kernel.
- **Malware exclusion** means that apps cannot download harmful code or files once installation is complete. As Guideline 2.5.3 states,[4] "[a]pps that transmit viruses, files, computer code, or programs that may harm or disrupt the normal operation

[4] See PX56.8.

of the operating system and/or hardware features . . . will be rejected." Note that malware exclusion is a different security goal than exploit resistance. Exploit resistance makes it harder for a well-intentioned but buggy app to be attacked; in contrast, malware exclusion prevents a developer from creating a benign-seeming app which passes the app review, but fetches intentionally-malicious content once installation is complete.

- **User consent for private data access** has three aspects: the initial providing of consent, the potential future revocation of the consent, and the secure storage and transmission of that data. There are two kinds of private data. **Computer-generated private data** comes from phone hardware like a camera, or from software-managed databases like a contact list. **User-generated private data** is provided to a phone by the phone's user; examples of such data include a user's birthday or Social Security number. If consent is given for access to either type of data, Apple requires apps to handle the data securely. In particular, Apple's review guidelines require sensitive data to be written to storage devices in encrypted form, and transmitted to remote servers in encrypted form.

- **Legal compliance** indicates that an app satisfies the laws belonging to a user's legal jurisdiction. For example, apps used by E.U. citizens should comply with General Data Protection Regulation (GDPR) requirements involving data privacy.

ii. *Developer Certification and App Submission*

51. Before submitting an app for review, a developer Alice must register with **Apple's Developer Program**. Alice uses her local computer to create a public key and a private key (as defined by Paragraph 40). Once Alice has created a public key and a private key, Alice contacts the Apple Developer Program, sending her public key and some additional information like her email address. Apple returns a **certificate** to Alice. A certificate is a signed statement which says "<person>'s public key is <a big number representing the public key>." Alice's certificate is signed by Apple, and says "Alice's public key is <whatever Alice's public key is>." Remember that a signature provides attribution and integrity (see Paragraph 40). So, Alice's certificate proves that Apple has bound Alice to a specific public key, such that no one can later modify the certificate to claim that Alice is bound to a different public key.

52. When Alice wants to submit her app for review, she first signs the app using her private key. Alice sends the signed app and her certificate to Apple. Apple verifies that her certificate was signed by Apple, ensuring that Alice has registered with Apple's Developer Program. If the certificate verification succeeds, Apple uses the public key in the certificate to verify the app signature. The verification of Alice's unforgeable signature allows Apple to determine that only Alice could have submitted the app; this accountability allows Apple to attribute malicious app submissions to Alice.

53. If Alice's app passes the review, Apple will sign the app, placing the Apple-signed version in the App Store. Since only Apple can generate Apple's signature, the signature

represents Apple's figurative "stamp of approval" for the app. *Figure 7* depicts the workflow for registering with the Developer Program and submitting an app for review.

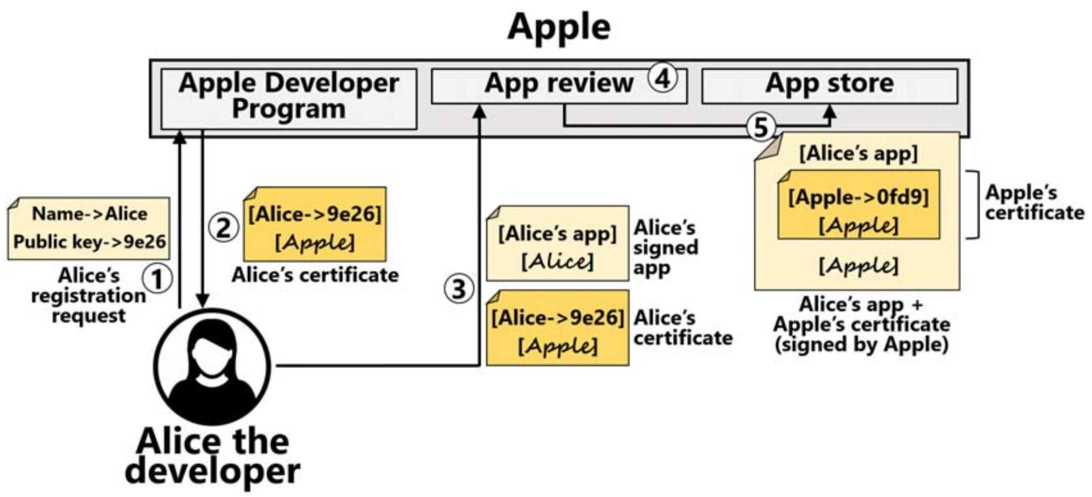

Figure 7: Example of how app submission works; cursive font indicates a signature. When Alice the developer wants to submit an app to the App Store, she first sends a registration request to the Apple Developer Program (①). The request contains Alice's name and her public key. Apple responds with a certificate for Alice (②). The certificate, which is signed by Apple, binds the developer name "Alice" to the public key 9e26. Once Alice has the certificate, she uses her private key to sign her app. She then submits her signed app and her certificate to Apple (③). Apple verifies that Apple signed Alice's certificate; this check allows Apple to verify that Alice is a registered developer. Apple then verifies that Alice signed the app, ensuring that Alice was the person who submitted the app for review. If these checks are successful, Apple reviews the app (④). If the app passes the review, then Apple uploads the app to the App Store (⑤). Note that the uploaded app contains Alice's app, Apple's certificate, and a signature covering the app and the certificate. The signature uses the Apple key 0fd9 mentioned in Apple's certificate.

iii. *Installing an App*

54. Later, when a user wants to install Alice's app, the user must first locate the app in the App Store. Then, the user's iPhone will download the app from the App Store server, and check the app's signature. iOS will refuse to install the app if there is no signature, or if the signature was not created by Apple. An app can only get Apple's signature by successfully passing the Apple-mandated review process.

55. On the client-side (i.e., on the iPhone, not the App Store server), the App Store is implemented by the kernel and process-based OS middleware (Paragraph 44). **At a high level, iOS uses several different middleware processes to fetch the app from the App Store, verify the app's signature, and (if verification succeeds) install the app on the local storage device. Later, when the user tries to run the app, the kernel (assisted by additional process-based**

middleware) checks the signature on the app, and only allows the app to run if the signature is valid (i.e., issued by Apple).

B. Developer Enterprise Program

56. Most iPhones are used as personal devices. For these devices, Apple forces app distribution to go through the Apple-controlled App Store. However, an iPhone can be configured for use as an **enterprise device**. Apple allows the associated enterprise to privately distribute apps that can only be installed by the enterprise's employees on employee devices. In this scenario, apps do not undergo the Apple-controlled review process described in Paragraph 48.[5] Instead, Charlie the enterprise developer (who works at Acme Widgets) can create apps that other Acme employees can download from Acme servers and install on Acme enterprise iPhones, without app review from Apple.

57. I now provide more detail about how enterprise app distribution works. When Charlie the enterprise developer wants to create an app, he registers for Apple's **Developer Enterprise program**.[6] If Apple accepts the registration request, Apple gives Charlie a special kind of certificate that allows Charlie to sign enterprise apps. Those apps can be distributed via a server operated by Charlie's enterprise. When Charlie uploads an app to the enterprise-operated distribution server, Charlie signs the uploaded app with the enterprise developer key that Apple bound to Charlie during registration. The Charlie-signed app (accompanied by Charlie's enterprise certificate) is what gets downloaded by enterprise users; this part of the distribution workflow differs from the one in *Figure 7* (where the developer-signed app is uploaded for review, but the Apple-signed app is distributed to users).

58. When an enterprise user tries to install Charlie's app, the workflow is mostly similar to the one described in Paragraph 55, with the following exceptions:

- The enterprise user downloads the app from the enterprise-operated server, not the Apple-operated App Store;
- When the app is installed, and subsequently launched, the iPhone checks whether the app has been signed by Charlie the enterprise developer, not by Apple. Note that the Charlie-signed app contains Charlie's Apple-signed enterprise certificate;

[5] See Bashan, A. and Bobrov, O. "Enterprise Apps: Bypassing the Gatekeeper." Check Point. 2016. https://www.blackhat.com/docs/asia-16/materials/asia-16-Bashan-Enterprise-Apps-Bypassing-The-iOS-Gatekeeper-wp.pdf. However, pursuant to the terms of the Apple Developer Enterprise Program License Agreement, Apple "reserves the right to review and approve, or reject, any Internal Use Application". *See Epic Games, Inc., vs. Apple Inc.*, Case. No. 3:20-CV-05640-YGR (N.D. Cal.) Declaration of M. Brent Byars, Exhibit O ¶ 6.2 (Dkt. No. 41-15).

[6] https://developer.apple.com/programs/enterprise/.

thus, a phone can verify that Charlie has successfully registered as an Apple-blessed enterprise developer.

Once these modified steps are complete, an enterprise employee's iPhone can run Charlie's enterprise app—an app that was not vetted by Apple. The reason is that **iPhones already possess the ability to decouple *the downloading of an app from the Apple-operated App Store* and *the verification of the signature on an app*.** In other words, iOS's signature verification does not intrinsically require the associated app to originate from the Apple-operated App Store. **The iOS kernel code and process-based middleware that implement signature checking are already capable of being configured to accept third-party signatures (like an enterprise signature) as trustworthy.**

C. Unsigned Apple Development Model

59. iOS can be configured to totally disable signature checking for all code installed and executed on an iPhone. This configuration option cannot be used on the iPhones that Apple distributes to consumers and enterprises. However, Apple allows this configuration option on the devices that are used by Apple's own engineers. A primary reason is that, by default, iOS checks signatures not only for user-facing apps, but for iOS's process-based middleware. When Apple's engineers are testing new middleware, the requirement to constantly sign each piece of new middleware code is burdensome, particularly if that code changes frequently. So, during the testing phase, engineers can force iOS not to perform signature checks; in this model for app distribution, an engineer creates an app, distributes it to her own iPhone, installs it, and launches it without the app being reviewed by Apple or signed by anyone.

60. The existence of this configuration option is relevant to the topic of whether iPhones could easily support third-party app stores. The Developer Enterprise program demonstrates that iPhones can natively run signed apps that were neither reviewed nor signed by Apple. Unsigned Apple development mode indicates that, if Apple wanted, Apple could not only allow non-reviewed apps to run on iPhones—Apple could also allow those apps to be unsigned. Note that I am not endorsing a phone design in which all code is unsigned. Instead, I am observing that **Apple's restrictive approach towards the consumer-facing App Store is more limiting than what Apple's own iOS platform natively allows.**

D. iOS Sandboxing

61. System calls (Paragraph 25) are the sole mechanism by which an app can interact with external entities like the OS, other processes, and hardware devices. Each iOS process, regardless of whether it was created by Apple or a third-party developer, is associated with a system call filter (Paragraph 26) named a **profile**. A profile enumerates a set of permitted system calls, and a set of permitted values that can be passed to the allowed system calls. For example, a profile can specify that a process cannot issue system calls to read any files from the storage device. Alternatively, a profile could specify that system calls can be used to read certain files, but not all files. Profiles can also restrict the hardware that a process can access; for example, a process might be able to access a phone's camera, but not its microphone. Apple's official documentation for profiles is mostly high-level, but additional details have been discovered via reverse engineering.

62. iOS runs many Apple-provided processes that implement process-based middleware (as defined by Paragraph 44). iOS associates each of those middleware processes with a predefined profile. For example, consider a middleware process that assists the kernel with wireless communication. iOS assigns a profile which allows the system calls that configure a phone's cellular radio. However, the profile does not allow the process to issue system calls that (say) access a phone's camera; camera access is unnecessary for a process whose job is to manage wireless communication. Note that **the Apple-provided App Store middleware is sandboxed; third-party app store middleware could be sandboxed in the same way, such that it would have no more privileges (and thus pose no more of a security risk) than the first-party app store**.

63. iOS assigns the same profile to all third-party (i.e., non-Apple) apps. This profile is called `container`, to evoke the metaphor that iOS places each third-party app into a restricted environment. The `container` profile disallows the invocation of many system calls, forcing an app to access the associated functionality indirectly (if at all) via OS middleware. For many of the system calls that *are* allowed, the `container` profile requires the app to possess the appropriate **entitlements**. As described by Apple, "an entitlement is a right or privilege that grants an [app] particular capabilities." For example:

- The `Developer.SensorKit.Reader` entitlement provides access to sensor hardware like a phone's accelerometer and light meter.
- The `Security.Network.Client` entitlement allows an app to establish a network connection with a remote server.
- The `Security.Personal-information.AddressBook` entitlement provides access to the contacts in a user's address book.

When Alice the developer makes an app, she creates not only the code and data in the app, but a list of desired entitlements. For example, suppose that Alice is building a social networking app; to communicate with the app's datacenter servers, the app will require the `Security.Network.Client` entitlement.

64. Once Alice has finished designing her app, she submits a signed version to Apple's review process (see Step ③ in *Figure 7*). The signed app contains the app code, the app data, the list of desired entitlements, and a signature covering the code, data, and entitlements. Apple reviews the app for compliance with Apple's guidelines. If the app passes the review, a user David can subsequently find the app on the App Store and install it. Later, when the app tries to issue a system call, iOS verifies that (1) the system call is permitted by the `container` profile, and (2) the app possesses the necessary entitlements. Note that, even if an app possesses an entitlement, iOS will prompt David the first time that the app tries to use the entitlement; David must give explicit consent for the app to use the entitlement.

65. Sandbox compliance is enforced by the iOS kernel. Note that **sandboxing occurs regardless of the app distribution model (App Store, Developer Enterprise, or unsigned Apple development)**.

E. Summary

66. iOS supports three different models.

- The App Store model, which is the only one available to most iPhone users, forces apps to be reviewed by Apple, signed by Apple, and then distributed by the Apple-controlled App Store, with iPhones only accepting apps that bear Apple's signature.

- The Developer Enterprise model allows businesses to distribute corporate apps directly to employees. In this model, apps are unreviewed by Apple, and are signed but by the enterprise developers, not by Apple.

- The unsigned Apple development model is only available to Apple engineers. This model allows an Apple engineer to distribute her unsigned, unreviewed apps to her iPhones.

Importantly, for all three app distribution models, iOS applies the same set of OS-based security protections. For example, apps are still isolated using sandboxing, and protected from subversion via ASLR and W^X memory. Thus, iOS's core security mechanisms are independent of whether apps are reviewed by Apple or signed by Apple as required by the App Store distribution model.

VIII. The Security Implications of Apple's Review Process

67. As described in Section VII, Apple's review process screens for various application characteristics, but only some of them involve security. The security-related properties involve five categories: **sandbox compliance, exploit resistance, malware exclusion, user consent for private data access, and legal compliance**. As I explain below, the first three properties (sandbox compliance, exploit resistance, and malware exclusion) can be entirely enforced by the OS, without assistance from a separate app review process. The fourth property (user consent for private data access) is partially enforceable by the OS alone. As I describe below, some kinds of user consent cannot be enforced by the OS. Furthermore, the last security property (legal compliance) is mostly unenforceable by the OS. However, and importantly, **the security properties that an OS cannot enforce are also difficult for the app review to enforce. In practice, this means that app review provides minimal additional benefits to the security guarantees that could be provided by the OS.**

68. **Sandbox compliance** restricts how an app can interact with external entities like the OS, the local hardware, and other apps. A primary mechanism for sandbox compliance is system call filtering (see Paragraph 26). Filters limit which system calls an app can invoke, and restrict the data that an app can pass to those system calls. Remember that system calls are the sole mechanism by which an app can interact with external entities like the OS, other processes, and hardware devices. Thus, system call restrictions are the most fundamental constraint on the damage that a misbehaving app can unleash. **Enforcing sandbox compliance is totally unrelated to the app review process; the OS only needs to know what kind of sandbox restrictions to apply to an app.** For example, once the OS knows what an app's system call

filter is, the OS simply consults that filter whenever the app tries to make a system call. **As discussed in Section VII, iOS sandboxes apps regardless of how those apps arrive on a user's device.**

69. Exploit resistance means that user-level code and kernel code are difficult to subvert.

- **Exploit resistance for user-level code:** A critical responsibility of an OS is to make the subversion of user-level applications more difficult. Even if a process's code has bugs, an OS should make it difficult for attackers to exploit those bugs. OSes have various techniques for conferring exploit resistance upon apps: ASLR (Paragraph 35), W^X (Paragraph 34), and PPL (Paragraph 37). As explained in Section X, **iOS can enforce all of these exploit resistance techniques regardless of whether an app has been reviewed by Apple.**

- **Exploit resistance for kernel code:** As explained in Paragraphs 36 and 37, techniques like KIP and PPL thwart attackers from modifying preexisting kernel code, injecting new kernel code, or leveraging bugs in preexisting kernel code to disable memory protections. **All of these security techniques are independent of an app review mechanism**, because the success of these techniques is only dependent on how the OS is structured, not how user-facing apps are structured.

70. **Malware exclusion** prevents an app from downloading harmful code or files once installation is complete. **Malware exclusion is independent of app review, because a computer can use anti-malware techniques regardless of how apps are delivered.** For example, anti-malware products like Norton 360, McAfee Total Protection, Avast Antivirus, and Windows Defender are able to scan an app-to-install regardless of the app's origins. The reason is that malware detection is primarily based on examining app content and app behavior. For example, a malware detector might examine an app's code to see if the code matches that of a known virus. A malware detector may also perform a "trial run" of the app, running it inside of a sandbox to observe the app's behavior in a way that keeps the rest of the computer safe. Apple's public-facing documentation does not reveal how malware is detected during the app review process; however, the techniques are likely similar to those employed by client-side anti-malware software and cloud-based "virus-scan-as-a-service" products.

71. **User consent for private data access** has three aspects: the initial providing of consent, the potential future revocation of the consent, and the secure storage and transmission of that data. There are two kinds of private data. **Computer-generated private data** comes from phone hardware like a camera, or from software-managed databases like a contact list. **User-generated private data** is provided to a phone by the phone's user; examples of such data include a user's birthday or Social Security number. If consent is given for access to either type of data, Apple requires apps to handle the data securely.

72. First consider computer-generated data.

- An app must ask a user for explicit consent before accessing hardware like the camera, or a sensitive database like a contact list. **An OS can enforce user**

consent for computer-generated data without help from an app store review. When an app wants to access such a hardware component or OS-managed resource, the app must issue a system call which clearly identifies the resource in question. Upon receiving such a system call, the OS can prompt the user, allowing the user to consent to the access. Apple's Guideline 2.5.14 requires that apps "provide a clear visual and/or audible indication when recording, logging, or otherwise making a record of user activity." Such indicators can be provided by the OS itself, without assistance from the app, since (1) the OS knows when an app uses system calls to access sensitive hardware or databases, and (2) the OS has ultimate control over the images the touchscreen displays and the sounds the speaker emits, and thus can display visual and/or audible indications when hardware access is detected. Also note that, because the app's entitlement list is covered by the signature, the app cannot successfully modify the list post-installation, e.g., to add more entitlements than the user originally consented to. Any such violation of entitlement integrity would be detected by the OS during the signature verification that occurs when the app is loaded.

- **An OS can also ensure that apps store and transmit user data safely.** OS-enforced memory isolation (Paragraph 24) prevents one app from reading or writing user data that lives in the memory region belonging to another app. To ensure that apps only write encrypted user data to storage devices, and only transmit encrypted user data to remote servers, the OS-provided I/O mechanisms (Paragraph 25) can provide encryption by default. For example, an OS can ensure that network communication with a web server always uses HTTPS (which is encrypted) instead of HTTP (which is not).

- If a user allows an app to access sensitive hardware or a sensitive database, then at any time in the future, a user must be able to revoke the app's access. The user could request the OS to disable any or all of the app's entitlements. (For example, the OS could implement this functionality via a configuration menu.) The OS could do this even if the app does not willingly provide an in-app mechanism to disable an entitlement. The reason is that apps exercise the right to an entitlement by making system calls; the OS determines whether such a system call should be allowed. An app cannot "undo" an OS-mandated disabling of an entitlement, because the OS keeps the list of disabled entitlement in storage space that the app cannot access. Thus, **for computer-generated private data, consent revocation is fully implementable by the OS alone, without assistance from an app store review.**

73. Now consider user-generated private data like a birthday or Social Security number. This kind of data is manually provided by a user via the general-purpose touchscreen, not by specialized phone hardware like a GPS unit, or by a special iOS database like the contact list. So, an OS cannot detect app access to (say) a Social Security number simply by seeing whether an app requests access to a particular hardware device or database; when an app receives touchscreen data, that data may or may not contain user-provided sensitive information. Furthermore, apps can request such information in a variety of formats using a variety of input

devices. For example, an app can request a birthday in month/day/year format, or day/month/year format, etc.; to collect that information, the OS can ask the user to input the data via the touchscreen, or verbally state the information for collection by a microphone. The net result is that iOS in particular (and mobile OSes in general) lack a generic way to detect which instances of user-submitted touchscreen data contain private information.

74. Apple's review guidelines suggest that Apple performs testing at review time and/or post-review to examine how apps collect and leverage user-submitted private data. For example, Guideline 5.1.1 says that "developers that use their apps to surreptitiously discover passwords or other private data will be removed from the Developer Program." Guideline 5.1.2 says that "apps that share user data [with third parties] without user consent or otherwise complying with data privacy laws may be removed." However, such testing is challenging to implement, since analysis of app behavior is difficult for apps of even moderate complexity.

75. Overall, on the topic of user consent for private data, my conclusion is that **an OS alone (without assistance from an app store review) can enforce user-consent restrictions for computer-generated private data. In contrast, an OS cannot enforce user-consent restrictions for user-generated private data, because there is no generic way that software like an OS can determine which user-generated data is sensitive data. Apple's Review Guidelines limit how apps should gather and share user-generated private data. However, empirical evidence shows that, in practice, Apple's review process (and some of Apple's own apps) enforce weak protections for user-generated private data.** For example, in May 2019, The Wall Street Journal tested 80 popular iOS apps, discovering that 79 contained third-party trackers.[7] A third-party tracker is a piece of code that an app includes for the purpose of sharing user information with parties who are neither the developer nor Apple. Data from third-party trackers is often sent to advertising companies, who analyze user data to deliver targeted ads to those users. As the Wall Street Journal said:

> "All but one [of the 80 apps] used third-party trackers for marketing, ads or analytics. The apps averaged four trackers apiece. Some apps send personal data without ever informing users in their privacy policies, others just use industry-accepted—though sometimes shady—ad-tracking methods . . . many apps send info to Facebook, even if you're not logged into its social networks. In our new testing, we found that many also send info to other companies, including Google and mobile marketers, for reasons that are not apparent to the end user."

76. **Legal compliance** ensures that an app satisfies the laws belonging to a user's legal jurisdiction. Determining an app's legal compliance is challenging, for the same reasons that understanding how an app handles user-generated private data is hard: except for a few cases like digital rights management, the desirable legal properties of an app are not crisply defined in a way that either software or a human can quickly and accurately verify. There is no obvious technical means by which Apple could write software to automatically check apps for

[7] Stern, J. "iPhone Privacy Is Broken…and Apps Are to Blame." Wall Street Journal. May 31, 2019. https://www.wsj.com/articles/iphone-privacy-is-brokenand-apps-are-to-blame-11559316401.

compliance with arbitrary laws. Importantly, many of the clearly relevant laws (e.g., the GDPR) involve the handling of user-generated private data; as described in Paragraph 75, Apple's review process struggles to accurately evaluate the ways in which apps handle such data. So, I conclude that **an OS alone cannot enforce legal compliance, but app store reviews provide enforcement that is weak at best.**

77. In summary, this section examined five security goals for the app store review: sandbox compliance, exploit resistance, malware exclusion, user consent for private data access, and legal compliance. The section demonstrated that an OS alone can fully enforce the first three properties, and partially enforce the fourth property. The security properties which an OS cannot enforce are also difficult for an app review to enforce. In practice, this means that an app review provides minimal additional benefit to the security guarantees that are already or could be provided by the OS. **It is therefore my expert opinion that Apple considerably overstates the security benefits of its centralized App Store model.**

78. As supporting evidence, consider the frequent Apple claim that iPhones (which use a curated app review and a centralized distribution channel) are more secure than Google's Android devices (which allow third-party app stores). A security evaluation of hundreds of iPhone apps from 2019 and 2020 found that iPhone apps suffered from the same security problems found in many Android apps.[8] For example, the security evaluation found that many iOS apps stored passwords on a phone's storage device without encrypting the passwords. Many of the iOS apps also transmitted sensitive information over the network in unencrypted form. All of these apps passed the Apple-mandated review process, but violated Apple's own expectations for secure handling of private user data (see Paragraph 71).

79. Apple's review process has also been criticized by app developers who believe that Apple enforces the review guidelines inconsistently. For example, a Verge article from 2021[9] describes complaints from multiple developers about **copycat apps**, **scam apps**, and **review fraud**. A copycat app is one that duplicates the primary functionality of a preexisting app. A scam app is one that charges exorbitant subscription fees, inflates the costs of in-app purchases, or deceives users into making unintended purchases. Both copycat apps and scam apps are specifically forbidden by Apple's review guidelines. Those guidelines state that developers should not "simply copy the latest popular app on the App Store, or make some minor changes to another app's name or UI and pass it off as your own"; the guidelines also state that Apple "won't distribute apps and in-app purchase items that are clear rip-offs. We'll reject expensive apps that try to cheat users with irrationally high prices". The Verge article explains how an app developer named Kosta Eleftheriou discovered that his app "was maliciously copied by numerous developers that built non-functioning versions of the software and charged

[8] See a high-level overview of the evaluation at Wagenseil, P. "iPhone apps just as unsafe as Android apps, says security researcher." Tom's Guide. August 11, 2020. https://www.tomsguide.com/news/bad-ios-android-apps-bowne-dc28.

[9] See Statt, N. "Apple's App Store is hosting multimillion-dollar scams, says this iOS developer." The Verge. February 8, 2021. https://www.theverge.com/2021/2/8/22272849/apple-app-store-scams-ios-fraud-reviews-ratings-flicktype.

egregious subscription fees, only getting away with it because of strong App Store reviews and high five-star ratings he claims are fake." As the Verge article states:

> The issues Eleftheriou is raising are interlocking ones all stemming from what he says are inconsistently enforced App Store rules and lazy moderation. It's not just apps that try to siphon money away from consumers under false pretenses using exploitative subscription services. It's also fake reviews and ratings that can be bought and a broken algorithmic ranking system that helps these apps float to the top and take the spot of genuine paid apps developed by small teams or singular developers, Eleftheriou says. Letting it perpetuate, he adds, is a platform that Apple does not actively police unless it's an issue that gains the attention of the media or involves one of Apple's current rivals like Facebook or Fortnite maker Epic Games . . . Scores of other developers have begun chiming in with their own experiences, too.

The article states that Eleftheriou believes a "competing app store" (i.e., a third-party app store) would force Apple to improve its first-party App Store.

80. The apps described in Paragraph 78 were poorly engineered, but they were not intentionally designed to be malicious. However, there have been various cases in which intentionally harmful iPhone apps managed to pass Apple's review process. For example, in 2019, Apple had to remove 17 apps from the App Store[10] because those apps tricked users' iPhones into clicking on web advertisements; the attacker profited from each click because the attacker hosted the web pages which contained the ads. This advertising model is known as "pay-per-click," and is used by non-malicious sites as well. The basic idea is that a web site publisher (say, CNN) allows advertisers to display ads on the web site; when a user clicks on an ad, the advertiser pays a fee to the web site publisher. As another example, in 2018, Apple had to remove two fitness apps because of financial fraud.[11] Both fitness apps requested users to provide their thumbprints through the Touch ID fingerprint scanning feature on iPhones, ostensibly to allow the apps to generate personalized exercise plans. However, the apps actually used the fingerprint scans to trick the iPhone into authorizing financial payments from the users to the app developers.

81. For another security comparison between Android and iOS, consider the market rate for **zero-day vulnerabilities**. A zero-day vulnerability is a security bug that is unknown to the developers of the buggy program. Attackers who discover such a vulnerability can exploit the vulnerability immediately, knowing with complete confidence that the attack will succeed. Such

[10] See Threat Post's coverage of the story at O'Donnell, L. "Apple Removes 17 Malicious iOS Apps From App Store." Threat Post. October 24, 2019. https://threatpost.com/click-fraud-malware-apple-app-store/149496/.

[11] See the We Live Security article at Stefanko, L. "Scam iOS apps promise fitness, steal money instead." We Live Security. December 3, 2018. https://www.welivesecurity.com/2018/12/03/scam-ios-apps-promise-fitness-steal-money-instead/.

a vulnerability is called a zero-day exploit because, when the developers of the buggy app learn about the security problem, the developers have "zero" days to fix it—the bug is already being used by real attackers, so a security fix is needed immediately.

82. There is an open market for zero-day bugs. Publicly-known companies (and hidden ones known only to criminals) bid on zero-day vulnerabilities that are created by hackers. In turn, the companies sell those vulnerabilities to the companies' customers, who might be nation-state governments, criminals, or anyone else with sufficient financial resources. Zerodium is one of the best-known resellers for zero-day exploits. Historically, Zerodium paid the highest bounties for exploits that targeted iOS devices. However, in 2019, Zerodium began paying its highest bounty for Android exploits.[12] As of December 2020, Zerodium still provides the biggest payout for Android exploits.[13] A reasonable interpretation of this market data is that iOS phones are now easier to compromise than Android phones, making iOS exploits less valuable. Indeed, this belief is held by Zerodium's CEO, who is quoted in a Wired article as saying the following:

> "During the last few months, we have observed an increase in the number of iOS exploits, mostly Safari and iMessage chains, being developed and sold by researchers from all around the world. The zero-day market is so flooded by iOS exploits that we've recently started refusing some them . . . [meanwhile,] Android security is improving with every new release of the OS thanks to the security teams of Google and Samsung, so it became very hard and time consuming to develop full chains of exploits for Android and it's even harder to develop zero-click exploits not requiring any user interaction."

83. Of course, market data must always be interpreted with a degree of skepticism. Another plausible interpretation for the falling price of iOS exploits is that companies like Zerodium traditionally reserved the largest bounties for iOS exploits; this higher level of demand led to a higher level of supply, which is now depressing the demand. Regardless, the existence of a robust market for both Android *and* iOS vulnerabilities suggests that Apple's App Review process does not lead to an overwhelmingly large security advantage.

IX. The macOS Operating System

84. macOS is the Apple-created operating system that runs on Apple's desktop and laptop devices. There are three ways that apps can be distributed on the macOS platform: (i) through the Mac App Store; (ii) through a third-party distribution channel with Apple's notarization; and (iii) through a third-party distribution channel without Apple's notarization.

[12] See Wired's coverage of this development at Greenberg, A. "Why 'Zero Day' Android Hacking Now Costs More Than iOS Attacks." Wired. September 3, 2019. https://www.wired.com/story/android-zero-day-more-than-ios-zerodium/.

[13] See the "ZERODIUM Payouts for Mobiles" chart at "Our Exploit Acquisition Program." Zerodium. https://zerodium.com/program.html.

85. First, users may install apps that are available via the Apple-managed **Mac App Store**. This store is analogous to the iOS store. Developers sign their apps and upload them to Apple for review; apps which pass the review are then made available to users via the app store. As described by Apple's security documentation for macOS developers,[14] "apps downloaded from the Mac App Store are always approved [by Apple] for use," in the same way that iOS always allows users to install apps distributed via the official iOS store. The justification is that these apps have passed Apple's review process. The client-side of the Mac App Store is implemented in a similar way to the client-side of the iOS App Store; a combination of kernel code and process-level middleware handle the downloading of apps, the checking of signatures, and so on.

86. Apple also allows macOS developers to distribute apps via third-party channels, e.g., a developer-operated server. However, if such a developer is registered with Apple's developer program (see Paragraph 51), Apple requires the developer to submit their app for **notarization**. App notarization is an automated process that is different than App Review. A developer first uploads her signed app to Apple's server, which then scans the app for malware and produces a statement, signed by Apple, which declares that the app is free from malware recognized by Apple. The developer attaches this notarization statement to her app; the notarized app consists of the app plus the statement. Later, when a user downloads the notarized app, macOS checks whether the notarization statement was signed by Apple; only if this is true will macOS run the app.

87. By default, macOS allows users to install (1) software from the App Store and (2) software that is notarized. Both kinds of software require developers to be registered with Apple. However, many third-party macOS developers do not officially register with Apple.[15] Unregistered developers lack an Apple-provided developer ID, and thus could not submit their apps for notarization even if the developers wanted such notarization. However, macOS allows users to install **unsigned, un-notarized third-party apps** that were not distributed via the Mac App Store, and were not submitted for notarization.[16] If the user tries to install such an app, macOS will display a warning dialog. The user can override the warning and decide to install the app, although Apple makes this decision cumbersome by forcing users to go into their security settings and manually enable the installation of such apps.

88. **For any given app, macOS enforces security mechanisms like ASLR, sandboxing, and W^X bits, regardless of how the app was distributed to a user's device.**

---

[14] https://developer.apple.com/design/human-interface-guidelines/macos/app-architecture/security/.

[15] For an example of a developer complaining about the friction of the MacOS App Store, see Jaffee, A. "Beyond the Sandbox: Signing and distributing macOS apps outside of the Mac App Store." AppCoda. May 29, 2020. https://www.appcoda.com/distribute-macos-apps/#Advantages_of_staying_out_of_the_Mac_App_Store.

[16] https://support.apple.com/guide/mac-help/open-a-mac-app-from-an-unidentified-developer-mh40616/mac.

Much like in iOS, those security mechanisms in macOS are implemented by kernel code and process-based middleware code that is agnostic about app distribution mechanisms.

X. Design Implications for Third-Party App Stores on iOS

89. Sections VII and IX gave an overview of iOS and macOS, describing how the two operating systems use a shared kernel, and a partially overlapping set of process-based middleware, to implement six models for app distribution. The table below summarizes whether each of the models requires apps to be reviewed by Apple, and/or bear a signature from some entity.

App Distribution Model	Apps Reviewed by Apple	Apps Signed By:
iOS: App Store	Yes	Apple
iOS: Developer Enterprise	No	Enterprise developer
iOS: Unsigned Apple Dev	No	Nobody
macOS: Mac Store	Yes	Apple
macOS: Notarized	Yes, but just a malware scan	Third-party developer
macOS: Unsigned Third-Party	No	Third-party developer or nobody

Figure 8: The various models for app distribution on iOS and macOS.

90. *Figure 8* induces several important observations. First, *Figure 8* demonstrates that iOS and macOS can support a diversity of app distribution models. This diversity is possible because **iOS and macOS are modular operating systems. The underlying kernel can be paired with various kinds of process-based middleware; different pairings enable different kinds of app distribution channels.**

91. A second observation is that **app distribution models which are possible on macOS are possible on iOS, and vice versa. The reason is that macOS and iOS share the same kernel. That kernel manages the installation of new apps, the launching of new app processes, and the invocation of system calls by app processes; the ability to execute security polices during those operations is the foundation for sandboxing and signature checking (if desired).** For example, from the engineering perspective, the client-side design of the macOS Mac Store middleware is similar to the client-side design of the iOS App Store middleware. As another example, implementing the notarization model on iOS would be straightforward, because iOS shares the same kernel as macOS, and that shared kernel provides the necessary introspection points to implement notarization.

92. A third observation is that, **in all six distribution models, OS-based security mechanisms like sandboxing and ASLR are still enforced. The reason is that those mechanisms are independent of the means by which an app arrives on a device. So, opening iOS to third-party app stores would not somehow disrupt iOS's preexisting security mechanisms.**

93. A final observation is that, **of the six preexisting distribution models, only two require apps to be fully vetted by Apple. A third essentially only requires apps to be**

screened by an Apple-run malware scanner. A reasonable conclusion is that opening iOS to third-party app stores would not significantly jeopardize the security of Apple devices, because Apple already trusts iOS and macOS, not a review process, to act as the primary guarantor of security.

94. Indeed, despite the fact that the iOS Developer Enterprise program allows a third-party business to distribute unreviewed apps to company employees, Apple still claims to provide "comprehensive methods [that] help protect corporate data in an enterprise environment."[17] That claim is true because, in practice, iPhone security is largely independent of how an app arrives on a device (and whether the app was reviewed before distribution). Instead, iPhone security is largely enforced by the client-side operating system.

95. Even though Apple allows macOS computers to download third-party apps directly from user-selected distribution channels, Apple still touts the security of the macOS experience too. Those security claims are consistent with my opinion that device security is largely enforced by the operating system. However, those claims are inconsistent with Apple's other statements which assert (wrongly, in my opinion) that third-party app stores are fundamentally incompatible with secure computing experiences.

96. Note that Apple allows iPhones to synchronize (i.e., share) data with macOS devices belonging to the same user. For example, a user can synchronize an iPhone's photos, such that, when the phone takes a new photo, that photo will automatically become accessible to a user's macOS desktop computer; any changes to the photo made by the desktop computer will become visible to the phone, and vice versa. If an iPhone is synchronized with a macOS device, the security of the synchronized data is practically determined by the more permissive app model of macOS, not the comparatively strict app model of iOS. The reason is that, once data arrives on the macOS device (either because the data was shared by the iPhone, or because the data was natively generated by the macOS device), the macOS app model is what decides the set of macOS apps that can access the data. The macOS apps which access synchronized data might not have passed through the Apple review process. Indeed, those apps might not even be notarized. However, Apple must already be comfortable with this status quo, because Apple itself implemented the synchronization mechanisms. Apple presumably allows iPhone data to flow to macOS devices because Apple knows that the data will still be protected by OS-level security mechanisms implemented by macOS. Those security mechanisms are enforced despite the fact that macOS supports app distribution models that are more permissive than those of iOS.

XI. Conclusions

97. Epic asked me to evaluate the degree to which iPhone security is independent of the means by which apps are reviewed and later distributed to iPhones. My expert opinion is that iPhone security is in fact significantly independent of the review process and the distribution channel (however they may be implemented). Thus, my expert opinion is that Apple considerably overstates the security benefits of its centralized App Store model. Apple is justified in caring about the security of its users; however, an iPhone's security guarantees are

[17] See PX461.6.

predominantly enforced by the iPhone's operating system, not by Apple's App Store and the associated review process.

98. In theory, Apple's review guidelines enumerate some security properties that cannot be enforced by an operating system alone. However, a variety of empirical evidence suggests that, in practice, the App Store does a weak (at best) job of enforcing these additional security properties. A variety of empirical evidence also suggests that Android security is similar to (or perhaps even better than) Apple security, despite Android's differing approach to third-party app distribution.

99. My overall conclusion is that if Apple allowed iPhone users to opt into third-party app distribution channels, those users would not suffer from a meaningfully less secure experience. The fact that iOS and macOS already support a variety of app distribution models, all of which still enforce OS-provided security mechanisms, suggests that Apple is technically capable of supporting third-party iOS app stores.

* * *

Pursuant to 28 U.S.C. § 1746, I declare under penalty of perjury that the foregoing is true and correct and that I executed this written direct testimony on April 20, 2021, in Boston, Massachusetts.

WORD COUNT: 13,724

[illegible]

James W. Mickens, Ph.D.